Concurrent programming: From theory to practice

Concurrent Algorithms 2019
Vasileios Trigonakis
Principal Member of Technical Staff
Oracle Labs, Zurich
From theory to practice

Theoretical (design)

Practical (design)

Practical (implementation)
From theory to practice

Theoretical (design)

- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

Practical (design)

Practical (implementation)

Design (pseudo-code)
From theory to practice

**Theoretical (design)**
- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

**Practical (design)**
- System models
  - shared memory
  - message passing
- Finite memory
- Practicality issues
  - re-usable objects
- Performance

**Practical (implementation)**

Design (pseudo-code)

Design (pseudo-code, prototype)
## From theory to practice

### Theoretical (design)
- Impossibilities
- Upper/Lower bounds
- Techniques
- System models
- Correctness proofs

### Practical (design)
- System models
  - shared memory
  - message passing
- **Finite memory**
- Practicality issues
  - re-usable objects
  - **Performance**

### Practical (implementation)
- **Hardware**
- Which atomic ops
- Memory consistency
- Cache coherence
- Locality
- **Performance**
- Scalability

<table>
<thead>
<tr>
<th>Design (pseudo-code)</th>
<th>Design (pseudo-code, prototype)</th>
<th>Implementation (code)</th>
</tr>
</thead>
</table>
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Large = slow
  - Medium = medium
  - Small = fast
Why do we use caching?

- Core freq: 2GHz = 0.5 ns / instr
- Core → Disk = ~ms
- Core → Memory = ~100ns
- Cache
  - Core → L3 = ~20ns
  - Core → L2 = ~7ns
  - Core → L1 = ~1ns
Typical server configurations

- **Intel Xeon**
  - 14 cores @ 2.4GHz
  - L1: 32KB
  - L2: 256KB
  - L3: 40MB
  - Memory: 256GB

- **AMD Opteron**
  - 18 cores @ 2.4GHz
  - L1: 64KB
  - L2: 512KB
  - L3: 20MB
  - Memory: 256GB
Experiment
Throughput of accessing some memory, depending on the memory size
Outline

- CPU caches
- **Cache coherence**
- Placement of data
- Graph processing: Concurrent data structures
Until ~2004: single-cores

- Core freq: 3+GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → L3
  - Core → L2
  - Core → L1
After ~2004: multi-cores

- Core freq: ~2GHz
- Core → Disk
- Core → Memory
- Cache
  - Core → shared L3
  - Core → L2
  - Core → L1
Multi-cores with private caches

Private

Core 0
L1
L2
L3
Memory
Disk

Core 1
L1
L2

= multiple copies
Cache coherence for consistency

Core 0 has X and Core 1

- wants to write on X
- wants to read X
- did Core 0 write or read X?
Cache coherence principles

- To perform a write
  - invalidate all readers, or
  - previous writer
- To perform a read
  - find the latest copy
Cache coherence with MESI

- A state diagram
- State (per cache line)
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data
The ultimate goal for scalability

- Possible states
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data

- Which state is our “favorite”?
The ultimate goal for scalability

- Possible states
  - **Modified**: the only dirty copy
  - **Exclusive**: the only clean copy
  - **Shared**: a clean copy
  - **Invalid**: useless data

= threads can keep the data close (L1 cache)
= faster
Experiment
The effects of false sharing
Outline

- CPU caches
- Cache coherence
- Placement of data
- Graph processing: Concurrent data structures
Uniformity vs. non-uniformity

- Typical **desktop** machine

- Typical **server** machine

= Uniform

= non-Uniform
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data

Memory

C

L1
1

L2
7

L3
20

C

L1

L2

L3

C

L1

L2

L3
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data
Latency (ns) to access data

Conclusion: we need to take care of locality
Experiment
The effects of locality
Experiment
The effects of locality

vtrigona $ ./test_locality -x0 -y1
Size: 8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 2
Number of threads: 2
Throughput       : 104.27 Mop/s

vtrigona $ ./test_locality -x0 -y10
Size: 8 counters = 1 cache lines
Thread 0 on core : 0
Thread 1 on core : 10
Number of threads: 2
Throughput       : 43.16 Mop/s
Outline

- CPU caches
- Cache coherence
- Placement of data
- **Graph processing**: Concurrent data structures
### Relational view

<table>
<thead>
<tr>
<th>Name</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>Breaking bad</td>
</tr>
<tr>
<td>Rachid</td>
<td>Dexter</td>
</tr>
<tr>
<td>Vasilis</td>
<td>Dexter</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>Similar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breaking bad</td>
<td>Dexter</td>
</tr>
<tr>
<td>Dexter</td>
<td>Breaking bad</td>
</tr>
</tbody>
</table>
Graph processing

Relational view

<table>
<thead>
<tr>
<th>Name</th>
<th>Likes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>Breaking bad</td>
</tr>
<tr>
<td>Rachid</td>
<td>Dexter</td>
</tr>
<tr>
<td>Vasilis</td>
<td>Dexter</td>
</tr>
</tbody>
</table>

Graph view

<table>
<thead>
<tr>
<th>Name</th>
<th>Similar</th>
</tr>
</thead>
<tbody>
<tr>
<td>Breaking bad</td>
<td>Dexter</td>
</tr>
<tr>
<td>Dexter</td>
<td>Breaking bad</td>
</tr>
</tbody>
</table>

Graphs keep the connections among entities materialized.
Graph analytics

• Graphs have been studied in Math for centuries
  • Since Euler’s “Seven Bridges of Königsberg”, 1736
• Repeatedly traverse your graph and calculate math properties
• Classic graph problems
  • Graph isomorphism
  • Travelling salesman’s problem
  • Max flow, min cut
  • …
• More recent developments
  • Pagerank
  • Infomap
Graph queries

- **Graph pattern matching**
  - Query graphs to find sub-graphs that match a pattern e.g., triangle counting
- **Essentially: SQL for graphs**
Graph queries

- **Graph pattern matching**
  - Query graphs to find sub-graphs that match a pattern e.g., triangle counting
  - Essentially: SQL for graphs
  - Example: Friends of my friends

```sql
SELECT p1, p3, COUNT(p2)
MATCH (p1)-[:friend]->(p2)->[:friend]->(p3),
  ! (p1)-[:friend]->(p3)
WHERE p1.country = p2.country
GROUP BY p1, p3
ORDER BY COUNT(p2) DESC
```

Graph processing frequently involves both analytics and queries
Dissecting a graph processing system with a focus on (concurrent) data structures
Dissecting a graph processing system
Preparing for a job interview
with a focus on (concurrent) data structures
Architecture of a graph processing system
Architecture of a graph processing system

Tons of other data and metadata to store
Graph

tmp graph structure

“Vasilis”, “Breaking bad”, :likes
“Rachid”, “Dexter”, :likes
“Vasilis”, “Dexter”, :likes
“Dexter”, “Breaking bad”, :similar
“Breaking bad”, “Dexter”, :similar

graph structure

user-ids - internal ids

Vasilis ➔ 0 0 ➔ Vasilis
Rachid ➔ 1 1 ➔ Rachid
Breaking bad ➔ 2 2 ➔ Breaking bad
Dexter ➔ 3 3 ➔ Dexter

labels

:likes, :people, :similar, ...

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X
**Graph**

**tmp graph structure**

- "Vasilis", "Breaking bad", :likes
- "Rachid", "Dexter", :likes
- "Vasilis", "Dexter", :likes
- "Dexter", "Breaking bad", :similar
- "Breaking bad", "Dexter", :similar

**graph structure**

```
Vasilis -> 0 0 -> Vasilis
Rachid  -> 1 1 -> Rachid
Breaking bad -> 2 2 -> Breaking bad
Dexter   -> 3 3 -> Dexter
```

**user-ids - internal ids**

- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**labels**

- :likes, :people, :similar, ...

**properties**

- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**lifetime management**

- number_of_references: X

**Runtime**

**indices / metadata**

```
< 300  >= 300
```

**buffer management**

- 1MB

**task / job scheduling**

Producers

Consumers

```
| 1 | 2 | 3 | 4 |
```

- :likes, :people, :similar, :male ...

- {people, male} → {2,4}

**renaming (ids)**

- used

---

48
### Graph

**Graph Structure**
- “Vasilis”, “Breaking bad”, :likes
- “Rachid”, “Dexter”, :likes
- “Vasilis”, “Dexter”, :likes
- “Dexter”, “Breaking bad”, :similar
- “Breaking bad”, “Dexter”, :similar

**User-Ids - Internal Ids**
- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**Labels**
- :likes, :people, :similar, ...

**Properties**
- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**Lifetime Management**
- number_of_references: X

### Runtime

**Indices / Metadata**
- Vasilis
- Breaking bad
- Rachid
- Dexter

**Buffer Management**
- Vasilis
- Breaking bad
- Rachid

**Task / Job Scheduling**
- Producers
- Consumers

**Distinct**
- Vasilis
- Rachid

**Limit (Top k)**
- Vasilis, 2
- Rachid, 1

**Operations**

**Group By / Join**
- Vasilis, Breaking bad
- Rachid, Dexter

**BFS**

**DFS**

**Renaming (Ids)**
- used
- used
- used
Graph

**tmp graph structure**
- “Vasilis”, “Breaking bad”, :likes
- “Rachid”, “Dexter”, :likes
- “Vasilis”, “Dexter”, :likes
- “Dexter”, “Breaking bad”, :similar
- “Breaking bad”, “Dexter”, :similar

**graph structure**

```
+----+----+
|    |    |
|    |    |
|    |    |
```

**user-ids - internal ids**
- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**labels**
- :likes, :people, :similar, ...

**properties**
- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**lifetime management**
- number_of_references: X
Graph

- tmp graph structure
  - append only
  - dynamic schema
  → segmented table

Graph structure

user-ids - internal ids

Vasilis → 0 0 → Vasilis
Rachid → 1 1 → Rachid
Breaking bad → 2 2 → Breaking bad
Dexter → 3 3 → Dexter

labels

:likes, :people, :similar, ...

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X
Graph

**tmp graph structure**
- “Vasilis”, “Breaking bad”, :likes
- “Rachid”, “Dexter”, :likes
- “Vasilis”, “Dexter”, :likes
- “Dexter”, “Breaking bad”, :similar
- “Breaking bad”, “Dexter”, :similar

**graph structure**

**user-ids - internal ids**
- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

**labels**
- :likes, :people, :similar, ...

**properties**
- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**lifetime management**
- number_of_references: X

- **tmp graph structure**
  - append only
  - dynamic schema
  → segmented table

- **Classic graph structures**
Graph

- tmp graph structure
  - append only
  - dynamic schema
  → segmented table

- Classic graph structures
  1. connectivity matrix

```
  0 1 2
0  x
1  x  x
2  x
```

  2. adjacency list

```
0 - 0
1 - 0 - 2
2 - 1
```

  3. compressed source row (CSR)

```
1 3 4
0 0 2 1
```

user-ids - internal ids

- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

labels

:likes, :people, :similar, ...

properties

- "Vasilis", {people, male}, 33, Zurich
- "Rachid", {people, male}, ??, Lausanne

lifetime management

number_of_references: X
• Mapping user ids to internal ids
  • create once
  • read-only after

user-ids - internal ids

Vasilis → 0
Rachid → 1
Breaking bad → 2
Dexter → 3

0 → Vasilis
1 → Rachid
2 → Breaking bad
3 → Dexter

labels
:likes, :people, :similar, …

properties
“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management
number_of_references: X
Graph

- Mapping user ids to internal ids
  - create once
  - read-only after
  \(\rightarrow\) hash map, lock-free reads

**tmp graph structure**

- "Vasilis", "Breaking bad", :likes
- "Rachid", "Dexter", :likes
- "Dexter", "Breaking bad", :similar
- "Breaking bad", "Dexter", :similar

**graph structure**

**CSR**

**user-ids - internal ids**

- Vasilis \(\rightarrow\) 0 \(\rightarrow\) Vasilis
- Rachid \(\rightarrow\) 1 \(\rightarrow\) Rachid
- Breaking bad \(\rightarrow\) 2 \(\rightarrow\) Breaking bad
- Dexter \(\rightarrow\) 3 \(\rightarrow\) Dexter

**labels**

- :likes, :people, :similar, ...

**properties**

- "Vasilis", {people, male}, 33, Zurich
- "Rachid", {people, male}, ??, Lausanne

**lifetime management**

- number_of_references: X
Graph

- Mapping user ids to internal ids
  - create once
  - read-only after
  → hash map, lock-free reads

- Mapping internal ids to user ids
  - create once
  - read-only after
  - fixed key range: [0, N]

user-ids - internal ids

<table>
<thead>
<tr>
<th>User</th>
<th>Id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Vasilis</td>
<td>0</td>
</tr>
<tr>
<td>Rachid</td>
<td>1</td>
</tr>
<tr>
<td>Breaking bad</td>
<td>2</td>
</tr>
<tr>
<td>Dexter</td>
<td>3</td>
</tr>
</tbody>
</table>

labels

:likes, :people, :similar, …

properties

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

lifetime management

number_of_references: X
Graph

- Mapping user ids to internal ids
  - create once
  - read-only after
  → hash map, lock-free reads

- Mapping internal ids to user ids
  - create once
  - read-only after
  - fixed key range: [0, N]
  → (sequential) array

```
user-ids - internal ids
Vasilis → 0  0 → Vasilis
Rachid → 1  1 → Rachid
Breaking bad → 2  2 → Breaking bad
Dexter → 3  3 → Dexter
```

```
labels
:likes, :people, :similar, ...
```

```
properties
“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne
```

```
lifetime management
number_of_references: X
```
Graph

- **Storing labels**
  - usually a small enumeration e.g., person, female, male
  - storing strings is expensive
    - “person” → ~ 7 bytes
  - comparing strings is expensive

**Graph structure**

- tmp graph structure
  - “Vasilis”, “Breaking bad”, :likes
  - “Rachid”, “Dexter”, :likes
  - “Dexter”, “Breaking bad”, :similar
  - “Breaking bad”, “Dexter”, :similar

- segmented buffer

- CSR

- user-ids - internal ids

  - Vasilis → 0
  - Rachid → 1
  - Breaking bad → 2
  - Dexter → 3
  - 0 → Vasilis
  - 1 → Rachid
  - 2 → Breaking bad
  - 3 → Dexter

- labels
  - :likes, :people, :similar, ...

- properties

  - “Vasilis”, {people, male}, 33, Zurich
  - “Rachid”, {people, male}, ??, Lausanne

- lifetime management

  - number_of_references: X
Graph

Storing labels
- usually a small enumeration e.g., person, female, male
- storing strings is expensive
  “person” → ~ 7 bytes
- comparing strings is expensive
  → dictionary encoding, e.g.,
  - person → 0
  - female → 1
  - male → 2

Ofc, hash map to
- store those
- translate during runtime

property

“Vasilis”, {people, male}, 33, Zurich
“Rachid”, {people, male}, ??, Lausanne

number_of_references: X
Graph

- Property
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges
- **Property**
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges → (sequential) arrays

- Graph structure
  - tmp graph structure
  - segmented buffer
  - graph structure
  - CSR

- User-ids - internal ids
  - Vasilis → 0
  - Rachid → 1
  - Breaking bad → 2
  - Dexter → 3
  - 0 → Vasilis
  - 1 → Rachid
  - 2 → Breaking bad
  - 3 → Dexter

- Labels
  - dictionary

- Properties
  - “Vasilis”, {people, male}, 33, Zurich
  - “Rachid”, {people, male}, ??, Lausanne

- Lifetime management
  - number_of_references: X
**Graph**

**tmp graph structure**

- “Vasilis”, “Breaking bad”, :likes
- “Rachid”, “Dexter”, :likes
- “Dexter”, “Breaking bad”, :similar
- “Breaking bad”, “Dexter”, :similar

**segmented buffer**

**graph structure**

- CSR

**user-ids - internal ids**

- Vasilis → 0
- Rachid → 1
- Breaking bad → 2
- Dexter → 3

- 0 → Vasilis
- 1 → Rachid
- 2 → Breaking bad
- 3 → Dexter

**hash map / array**

**labels**

- :likes, :people, :similar, ...

**dictionary**

**properties**

- “Vasilis”, {people, male}, 33, Zurich
- “Rachid”, {people, male}, ??, Lausanne

**lifetime management**

- number_of_references: X

**Property**

- one type per property, e.g., int
- 1:1 mapping with vertices/edges

→ (sequential) arrays

**Lifetime management**

(and other counters)

- cache coherence: atomic counters can be expensive
**Graph**

- **Property**
  - one type per property, e.g., int
  - 1:1 mapping with vertices/edges

→ (sequential) arrays

- **Lifetime management**
  - (and other counters)

- **Two potential solutions**
  1. **approximate counters**
  2. **stripped counters**

```
increment(int by) {    counter[my_thread_id] += by; }  
int value() {
    int sum = 0;
    for (int i = 0; i < num_threads; i++) { sum += counter[i]; }
    return sum;
}
```
Graph

tmp graph structure

```
"Vasilis", "Breaking bad", :likes
"Rachid", "Dexter", :likes
"Dexter", "Breaking bad", :similar
"Breaking bad", "Dexter", :similar
```

graph structure

```
CSR
```

user-ids - internal ids

```
Vasilis \rightarrow 0
Rachid \rightarrow 1
Breaking bad \rightarrow 2
Dexter \rightarrow 3
0 \rightarrow Vasilis
1 \rightarrow Rachid
2 \rightarrow Breaking bad
3 \rightarrow Dexter
```

hash map / array

labels

dictionary (= map)

properties

```
"Vasilis", {people, male}, 33, Zurich
"Rachid", {people, male}, ??, Lausanne
```

lifetime management

number_of_references: X

score

```
Structure           # Usages
array / buffer      5
map                 2
```

stripped counter
- **Indices**
  - Used for speeding up “queries”
  - Which vertices have label :person?
  - Which edges have value > 1000?

- **Labels**
  - :likes, :people, :similar, :male …
  - \{people, male\} → \{2,4\}
• Indices
  • Used for speeding up “queries”
    • Which vertices have label :person?
    • Which edges have value > 1000?
  → maps, trees
• Indices
  • Used for speeding up “queries”
    • Which vertices have label :person?
    • Which edges have value > 1000?

  → maps, trees

• Buffer management
  • In “real” systems, resource management is very important
  • buffer pools
    • no order
    • insertions and deletions
    • no keys

• Buffer management
  • In “real” systems, resource management is very important
  • buffer pools
    • no order
    • insertions and deletions
    • no keys
### Indices
- Used for speeding up “queries”
  - Which vertices have label :person?
  - Which edges have value > 1000?

→ maps, trees

### Buffer management
- In “real” systems, resource management is very important
- buffer pools
  - no order
  - insertions and deletions
  - no keys

→ Fixed num object pool: array
→ Otherwise: list
→ Variable-sized elements: heap
• Task and job scheduling
  • producers create and share tasks
  • consumers get and handle tasks
  • insertions and deletions
  • usually FIFO requirements
- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements
  \( \rightarrow \) queues

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
    e.g., \{person, female\}, \{person, male\}
- Task and job scheduling
  - producers create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements
  → queues

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
e.g., \{person, female\}, \{person, male\}
→ 2-level **dictionary** encoding
  - \{person, female\} → 0
  - \{person, male\} → 1
### Task and job scheduling
- producers create and share tasks
- consumers get and handle tasks
- insertions and deletions
- usually FIFO requirements

→ queues

### Storing / querying sets of labels
- set equality expensive
- usually common groups
e.g., \{person, female\}, \{person, male\}

→ 2-level **dictionary** encoding
- \{person, female\} → 0
- \{person, male\} → 1

### Giving unique ids (renaming)
- Task and job scheduling
  - produces create and share tasks
  - consumers get and handle tasks
  - insertions and deletions
  - usually FIFO requirements
    → queues

- Storing / querying sets of labels
  - set equality expensive
  - usually common groups
    e.g., \{person, female\}, \{person, male\}
    → 2-level **dictionary** encoding
    - \{person, female\} → 0
    - \{person, male\} → 1

- Giving unique ids (renaming)
  → tree, map, set, counter, other?
**Runtime**

- **indices / metadata**
  - map / tree
  - $< 300$, $\geq 300$

- **buffer management**
  - array

- **task / job scheduling**
  - queue
  - Producers → Consumers
  - task

- **labels**
  - :likes, :people, :similar, :male …
  - dictionary (= map)
    - \{people, male\} $\rightarrow$ \{2,4\}

- **renaming (ids)**
  - map / tree / set

---

**Score**

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>6</td>
</tr>
<tr>
<td>map</td>
<td>5</td>
</tr>
<tr>
<td>tree / heap</td>
<td>2</td>
</tr>
<tr>
<td>set</td>
<td>1</td>
</tr>
<tr>
<td>queue</td>
<td>1</td>
</tr>
</tbody>
</table>
- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- insertion only
- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- **insertion only**
  - hash map
  - atomic inc / sum / max, etc.

### Operations

**group by / join**

- Vasilis, Breaking bad
- Rachid, Dexter
- Vasilis, Dexter

- Vasilis, 2
- Rachid, 1

**distinct**

- Vasilis
- Rachid
- Vasilis

- Vasilis
- Rachid

**limit (top k)**

- 11 12 0 9 8 13
- 8 9 11 23 32 9
- 1 2 3 5 7 3 2 0

- 32
- 23
- 13

**BFS**

**DFS**
- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- **Join**
  - create a map of the small table
  - insertion phase, followed by probing phase
**Operations**

**group by / join**
- Vasilis, Breaking bad
- Rachid, Dexter
- Vasilis, Dexter

**distinct**
- Vasilis
- Rachid
- Vasilis

**limit (top k)**
- 11 12 0 9 8 13
- 8 9 11 23 32 9
- 1 2 3 5 7 3 2 0

**BFS**

**DFS**

- **Group by**
  1. Mapping from keys to values
  2. Atomic value aggregations e.g., COUNT, SUM, MAX
- insertion only
  → **hash map**
  → **atomic inc / sum / max**, etc.

- **Join**
  - create a map of the small table
  - insertion phase, followed by
  - probing phase
  → **hash map, lock-free probing**
• Distinct
  • can be solved with sorting, or

- Distinct

map / atomics

distinct

Vasilis
Rachid
Vasilis

limit (top k)

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

11 12 0 9 8 13
8 9 11 23 32 9
1 2 3 5 7 3 2 0

BFS

DFS

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, 1
Vasilis, 2
Rachid, 1
Vasilis, Dexter
- **Distinct**
  - can be solved with sorting, or
  - hash set
- **Distinct**
  - can be solved with sorting, or → **hash set**

- **Limit (top k)**
  - can be solved with sorting, or
  - different specialized structures
- **Distinct**
  - can be solved with sorting, or
    - **hash set**

- **Limit (top k)**
  - can be solved with sorting, or
  - different specialized structures
    - **tree**
    - **heap**
    - **~ list**
    - **array** (e.g., 2 elements only)
    - **register** (1 element only)
- Breadth-first search (BFS)
  - FIFO order
  - track visited vertices

### Operations

- group by / join
- map / atomics
- distinct
- hash set
- limit (top k)
- tree / heap / list

#### BFS

```
11 12 0 9 8 13
8 9 10 23 9
1 2 3 5 7 3 2 0
```

#### DFS

```
11 12 0 9 8 13
8 9 10 23 9
1 2 3 5 7 3 2 0
```

Vasilis, Breaking bad
Rachid, Dexter
Vasilis, Dexter
Vasilis, 2
Vasilis
Rachid
Vasilis
Rachid
Vasilis
Rachid
- **Breadth-first search (BFS)**
  - FIFO order
  - track visited vertices
  → queue
  → set

<table>
<thead>
<tr>
<th>Operations</th>
<th>BFS</th>
<th>DFS</th>
</tr>
</thead>
<tbody>
<tr>
<td>group by / join</td>
<td></td>
<td></td>
</tr>
<tr>
<td>map / atomics</td>
<td></td>
<td></td>
</tr>
<tr>
<td>distinct</td>
<td></td>
<td></td>
</tr>
<tr>
<td>hash set</td>
<td></td>
<td></td>
</tr>
<tr>
<td>limit (top k)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>tree / heap / list</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Vasilis, Breaking bad**
- **Rachid, Dexter**
- **Vasilis, 2**
- **Rachid, 1**
- **Vasilis, 11 12 0 9 8 13**
- **Rachid, 32 23 32 9**
- **Vasilis, 1 2 3 5 7 3 2 0**
- **Rachid, 4 3**
• Breadth-first search (BFS)
  • FIFO order
  • track visited vertices
  → queue
  → set

• Depth-first search (DFS)
  • LIFO order
  • track visited vertices
**Operations**

- **group by / join**
  - Vasilis, Breaking bad
  - Rachid, Dexter
  - Vasilis, Dexter

- **map / atomics**
  - Vasilis
  - Rachid
  - Vasilis

- **distinct**
  - Vasilis
  - Rachid
  - Vasilis

- **hash set**
  - Vasilis
  - Rachid

- **limit (top k)**
  - 11 12 0 9 8 13
  - 8 9 1 2 3 5 7 3 2 0

- **tree / heap / list**
  - 1
  - 3

**Breadth-first search (BFS)**
- FIFO order
- track visited vertices
  - queue
  - set

**Depth-first search (DFS)**
- LIFO order
- track visited vertices
  - stack
  - set
Operations

**group by / join**
- Vasilis, Breaking bad
- Rachid, Dexter
- Vasilis, 2

**map / atomics**
- Vasilis
- Rachid
- Vasilis

**distinct**
- Vasilis
- Rachid
- Vasilis

**limit (top k)**
- 11 12 0 9 8 13
- 32 23 0

**tree / heap / list**
- 8 9 11 23 32 9
- 1 2 3 5 7 3 2 0

**BFS**
- queue / set

**DFS**
- stack / set

---

**Score**

<table>
<thead>
<tr>
<th>Structure</th>
<th># Usages</th>
</tr>
</thead>
<tbody>
<tr>
<td>array / buffer</td>
<td>7</td>
</tr>
<tr>
<td>map</td>
<td>6</td>
</tr>
<tr>
<td>set</td>
<td>4</td>
</tr>
<tr>
<td>tree / heap</td>
<td>3</td>
</tr>
<tr>
<td>queue</td>
<td>2</td>
</tr>
<tr>
<td>stack</td>
<td>1</td>
</tr>
<tr>
<td>list</td>
<td>1</td>
</tr>
</tbody>
</table>
**Graph**

- **User-ids - internal ids**
  - Vasilis $\rightarrow$ 0
  - Rachid $\rightarrow$ 1
  - Breaking bad $\rightarrow$ 2
  - Dexter $\rightarrow$ 3
- **Hash map - array**
- **CSR**
- **Segmented buffer**

**Runtime**

- **Indices / metadata**
- **Map / tree**
- **Buffer management**
- **Task / job scheduling**
- **Group by / join**
- **Group / atomics**
- **Hash set**
- **Distinct**
- **Limit (top k)**
- **Tree / heap / list**

**Operations**

- **BFS**
- **DFS**
- **Queue / set**
- **Stack / set**

**Elements**

- **Labels**
  - :likes
  - :people
  - :similar
  - :male
- **Properties**
  - "Vasilis", {people, male}, 33, Zurich
  - "Rachid", {people, male}, ??, Lausanne
- **Lifetime management**
  - Number of references: X
  - Stripped counter

**Buffer Management**

- Segmented buffer
- Hash map / array
- CSR

**Dictionary**

- Producers
  - Vasilis
- Consumers
  - Rachid

**Queue**

- Producers
  - Vasilis
- Consumers
  - Rachid

**Map / Tree / Set**

- Producers
  - Vasilis
- Consumers
  - Rachid
Conclusions

• Both theory and practice are necessary for
  • Designing, and
  • Implementing fast / scalable data structures
• Hardware plays a huge role on implementations
  • How and which memory access patterns to use
• (Concurrent) Data structures
  • The backbone of every system
  • An “open” and challenging area or research

vasileios.trigonakis@oracle.com – internships++

Don’t forget: Oracle Labs event, this Wednesday